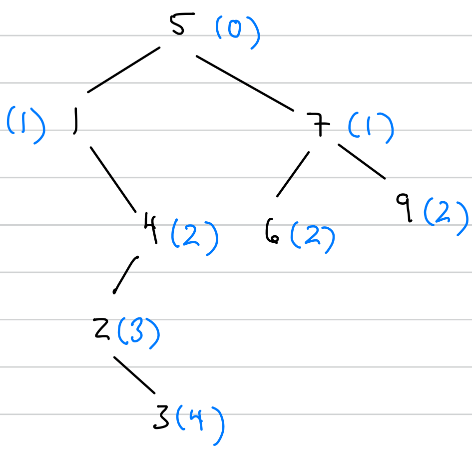
Assignment 3 solutions

1a) # of comparisons in blue next to each node.



Total # of comparisons:
$(1+2+3+4)+(1+2+2) = 15$

QuickSort on 5, 1, 4, 2, 7, 9, 3, 6 using pivots 5, 1, 4, 2, 3, 7, 6, 9

Partition about 5: $\underbrace{1\ 4\ 2\ 3}_{\text{pivot}=1}$ 5 $\underbrace{7\ 9\ 6}_{\text{pivot}=7}$. # comparisons: 7

Pivots 1 and 7: 1 $\underbrace{4\ 2\ 3}_{\text{pivot}=4}$ 5 6 7 9 # comparisons = 3 + 2 = 5

Pivot 4: 1 $\underbrace{2\ 3}_{\text{pivot}=2}$ 4 # comparisons. = 2.

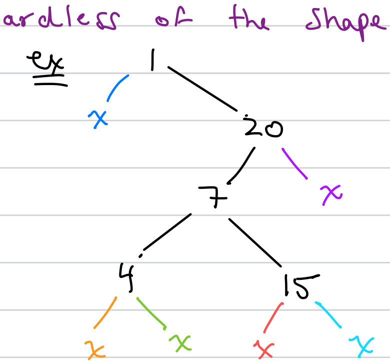
Pivot 2: 1 2 3 4 5 6 7 9 # comparisons: 1

∴ total # of comparisons $7+5+2+1 = 15$.

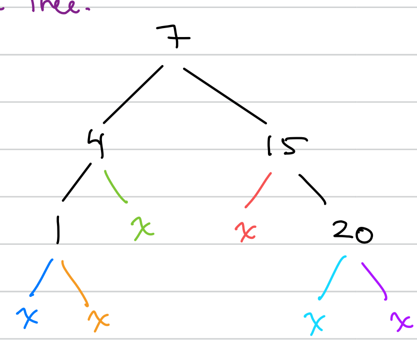
b) When n-1 elements have been inserted into the BST, the elements create exactly n intervals.

ex Tree w/ elements 1 4 7 15 20 defines intervals
$x<1$, $1<x<4$, $4<x<7$, $7<x<15$, $15<x<20$, $x>20$.
Each interval corresponds to exactly ONE insertion position in the tree, regardless of the shape of the tree.

ex



OR



∴ with n-1 elements in the tree, there are n intervals. The last element could go into any of those intervals.

∴ n possible insert spots.

c) PrintTree1: 50, 25, 12, 6, 35, 30, 40, 80, 70, 60, 95.

Prints out all nodes which have a left child. The order is based on preorder traversal.

PrintTree2: 50, 25, 12, 6, 35, 30, 40, 80, 70, 60,

Prints all nodes which have a left child, as long as all of its ancestors had a left child.

PrintTree 3: 50,

Prints all nodes that have a path of length ≥ 4 down the left-most branch of the subtree.

PrintTree 4: 0, 1, 2, 3, 4, 5.

Prints the height of each subtree along the left-most branch of the tree.

(d) Step 1) Create array A of size 2n

Use Inorder(T_1) and store sorted elements in array A.

Use Inorder(T_2) and concatenate elements to array A. } O(n)

Step 2) Use merge algorithm to combine the 2 sorted lists into one. } O(n)

Step 3) Use Build BST (A, 1, n) from class to Recursively build a BST using a sorted list. Recall the output will have all levels full except perhaps the last. O(n).

Step 4) Use ColorTree(T) from class which assigns colors to the tree from step 3. Recall all nodes are black, except those in last level, which are red. O(n).

Total time: O(n)

Q2 a) Solution below assumes initial input is not NIL.

TrimTree (T)

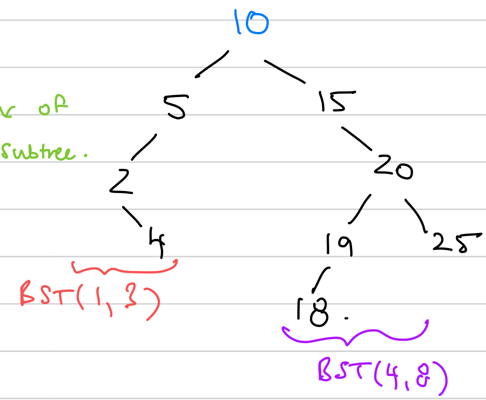
```
if T.left = T.right = NIL        // leaf
    return T  // don't delete T
if T.left = NIL     // node has 1 child => delete this node.
    R = TrimTree(T.right)  // Rec. call to right
    R.parent = T.parent      // update parent
    Return R          // delete node T
else if T.right = NIL
    L = TrimTree(T.left)  // Rec. call to left
    L.parent = T.parent    // update parent
    Return L.      // delete node T
else                       // Two Children.
    T.left = TrimTree(T.left)        // Recursively trim subtrees.
    T.right = TrimTree(T.right)
    Return T                // do not delete T
```

Runtime Recurrence: (worst case, make 2 rec. calls)

$T(n) = \underbrace{T(L_n) + T(R_n)}_{\text{Case of 2 children}} + c.$ $\therefore$ $T(n)$ is $O(n)$ since this is the recurrence of Inorder.

(b) RecreateBST(A, s, f)

```
if s ≤ f
    T = new Node()          // first item in array becomes root.
    T.Key = A[f]
    K = s
    while K ≤ f-1 and A[K] < T.Key    // find the last index of
        K = K+1                       elements belonging to left subtree.
    T.left = RecreateBST(A, s, K-1)   } Build
    T.right = RecreateBST(A, K, f-1)  } L/R subtrees.
    if T.left ≠ NIL
            T.left.parent = T         } Set parent
    if T.right ≠ NIL                  } Pointers
            T.right.parent = T        }
    return T.
else return NIL
```



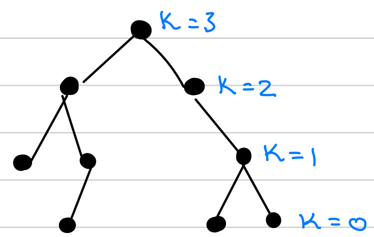
4 2 5 18 19 25 20 15 10

c) PrintDeepest(T)

```
    h = FindHeight(T)          // alg. from class, runs in O(n)
    PrintNodesatDepth(T, h)    // print all nodes at depth h.
```

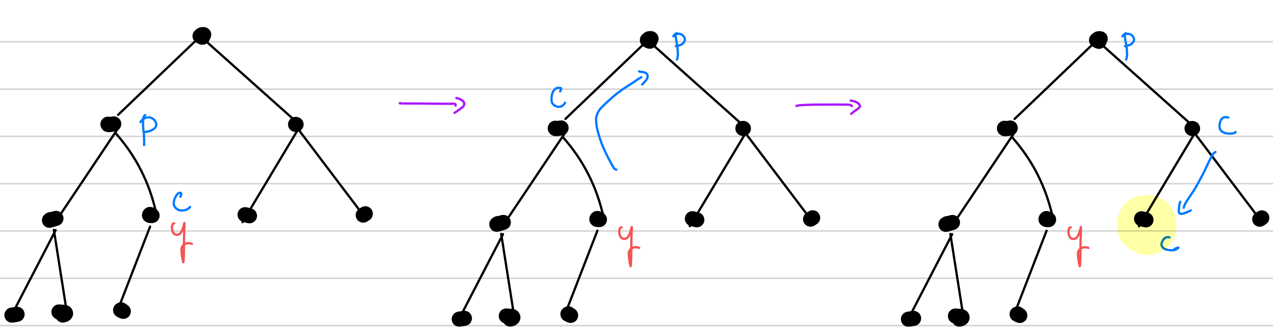
PrintNodesatDepth(T, K)

```
    if T ≠ NIL
        if K=0
            print T.Key
        else
            PrintNodesatDepth(T.left, K-1)
            PrintNodesatDepth(T.right, K-1)
```



$T(n) = T(L_n) + T(R_n) + c$ $\therefore$ $O(n)$, same as Inorder.

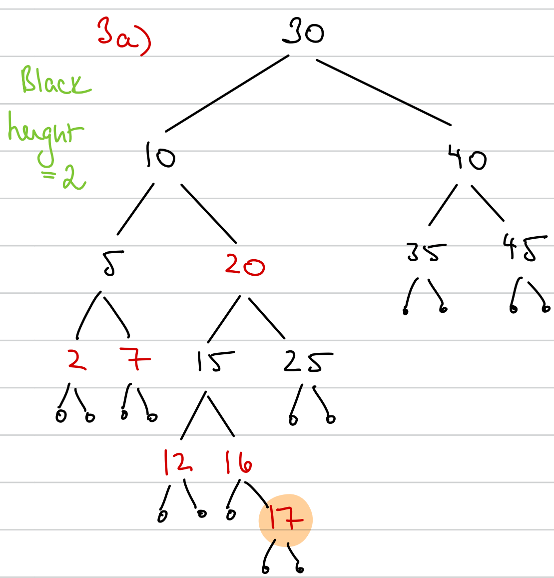
(d)

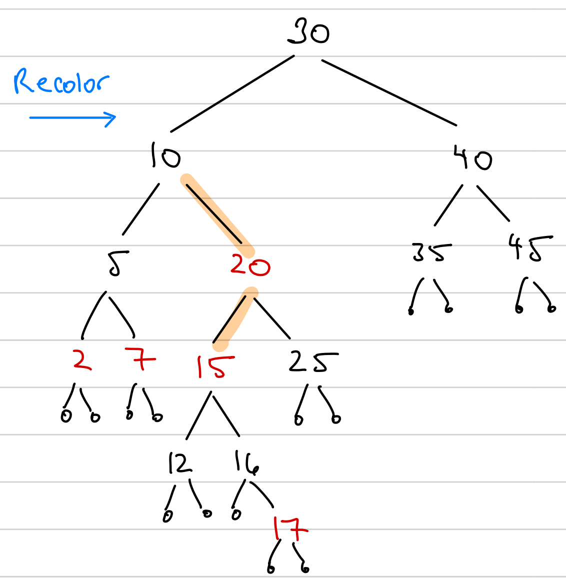


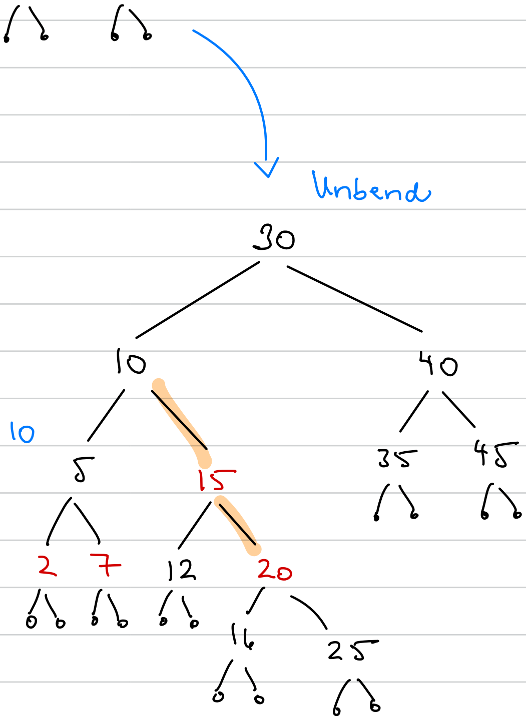
My Neighbour (T, y)

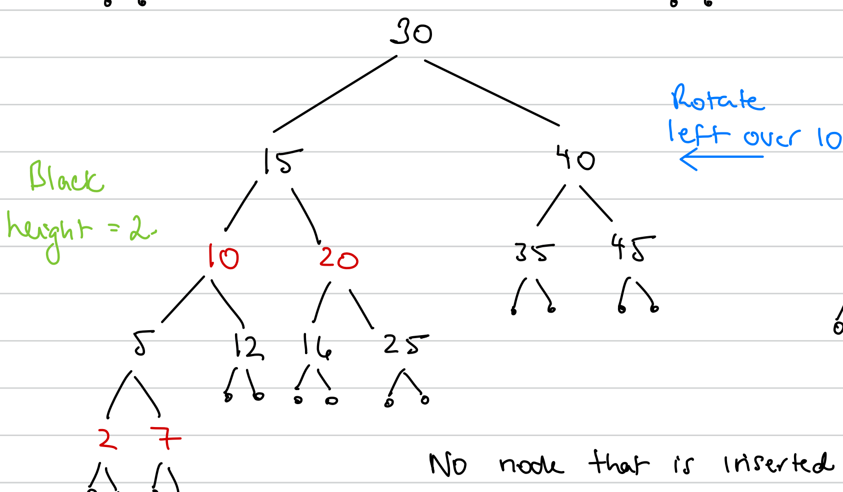
```
C = y                    // child ref that loops up tree
P = y.parent             // Parent ref that loop up tree.
count = 0                // count length of path until find a left child
while p.left ≠ c         // start a search up tree until you find a left child
    c = p
    p = p.parent
    count ++

c = p.right              // Now traverse to the right child of p.
while count > 0          // Search down the left path exactly "count" times.
    c = c.left
    count --
Return c
```

The alg. traverses up one path of the tree, and then down another, performing a constant amount of work at each step.
$\therefore$ Runtime is $O(h)$.









No node that is inserted next will change the bh. This only happens if the root is recolored, which would require the children of the root to be Red.

3b) (1) $n \le 2^{2b} - 1 \quad \therefore \quad n \le 2^{8} - 1 = 255$

(2) if $n = 45$, the tree with smallest black height must satisfy:

$2^{b} - 1 \le 45 \le 2^{2b} - 1$

$b = 2 \quad 2^{2} - 1 \le 45 \not\le 2^{2b} - 1$ Not Possible.

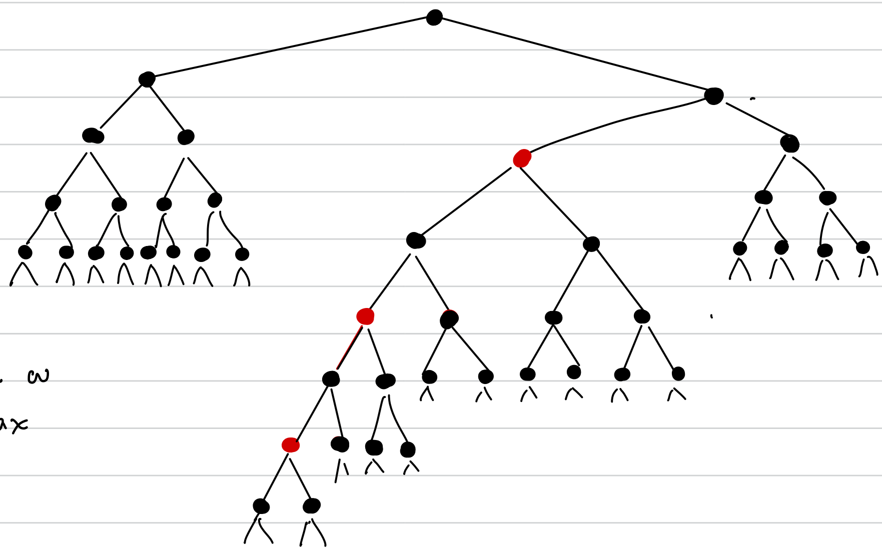
$b = 3 \quad 2^{3} - 1 \le 45 \le 2^{6} - 1$ Possible.

$\therefore$ min black height is 3.

(3) Note $b = 3, 4, 5$ are possible for $n = 45$:

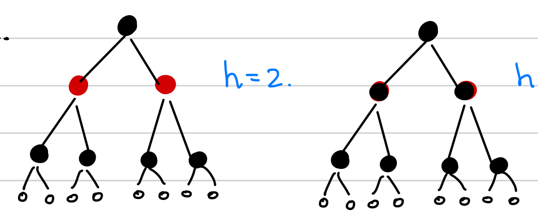
$2^{5} - 1 \le 45 \le 2^{10} - 1$

For $b = 5$, on the right we have the min # of nodes required for a bh of 5. We need 45 nodes to create a tree of height 7. We can't build a tree of $h = 8$ and $b = 5$. The max height is 7



(4) A RBT can only be colored all black if it is full and complete. There trees have 3, 7, 15, 31, 63, ... nodes. $(n = 2^{k} - 1)$. Since $45 \ne 2^{k} - 1$, it is impossible to have a full and complete tree (all black) on $n = 45$ nodes.

(5) More than one coloring is possible.



h=2. h=2.

3c) Recall $n \le 2^{2b} - 1$

$n + 1 \le 2^{2b}$.

$\underbrace{\frac{\log_2(n+1)}{2}}_{\text{min black height}} \le b$.

$n \ge 2^{b} - 1$

$n + 1 \ge 2^{b}$

$\underbrace{\log_2(n+1)}_{\text{max black heights.}} \ge b$.

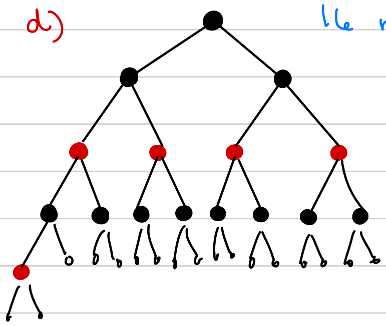
$\therefore$ max black height on n nodes is $\log_2(n+1)$. Therefore max height is

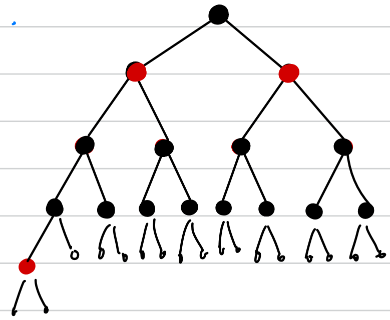
$2b - 1 = 2\log_2(n+1) - 1$.

$\therefore h$ is $O(\log n)$.

If $n = 100$, black height $\leq \log_2(101) = 6.65$. $\therefore$ black height of 15 is not possible.

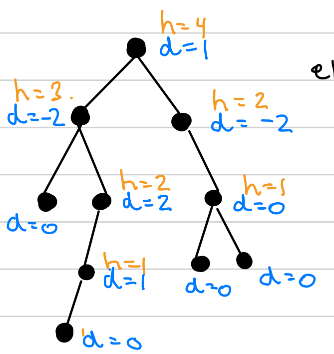
d) 16 nodes, $b = 3$.





e) $h = O(\log n)$ for RBT. We can carry out an insert in time $O(h) = O(\log n)$. Therefore we can build a RBT by inserting n times, which is $O(n \log n)$.

Q4 a)

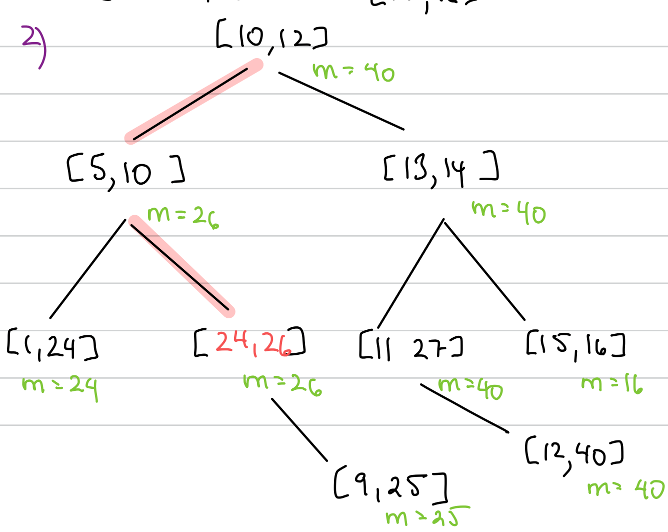



```
SetDiff( T)
      if T = NIL
            return -1
      else
            h1 = SetDiff (T.left) + 1     // max path length in left tree
            h2 = SetDiff (T.right) + 1    // max path length in right tree.
            d = h1 - h2.                  // difference in heights.
            T.diff = d.
            Return max(h1, h2)            // Return the height of this tree.
```

Recurrence: $T(n) = T(L_n) + T(R_n) + c$, same recurrence as Inorder. $\therefore O(n)$.

(C) 1) IntervalSearch(T, i) will search RIGHT if T.left.max < i.low. However, T.left.max ≥ 25. For i to overlap with [15, 16], it must be that i.low ≤ 16. Therefore it is impossible for Interval Search to return the blue [15, 16] node.

2)



$i = [25, 26]$

$25 \leq 26 \Rightarrow$ GO LEFT

$25 \not\leq 24 \Rightarrow$ GO RIGHT

Found overlap!

(d) PrintAll(T, i)

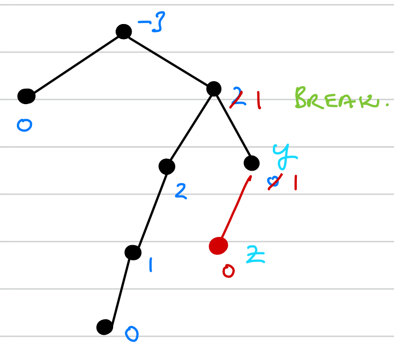
```
if T ≠ NIL
    if i overlaps with T.int        // found overlap.
            Print T.int
    if T.left ≠ NIL
        if i.low ≤ T.left.max       // intervals exist in left subtree
                Print All ( T.left)
    PrintAll (T.right)              // look for overlaps in right subtree.
```

(e) Tree operations on an interval tree run in time $O(h)$. ∴ implement the interval tree as a red black tree, where $h = O(\log n)$. With this implementation, inserts / deletes run in time $O(\log n)$.

b) Tree Insert (T, z)

```
z.diff = 0            // set diff of new node.
if T = NIL return z
else x = T
While x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if z.key < y.key
    y.left = z
else y.right = z
```

insert z into the tree as usual




```
while y ≠ T                // y is z's parent
    if z.key < y.key       // left child
        y.diff ++              // left branch increased
        if y.diff ≤ 0
            break          // break because right branch is ≥0 left branch
    else                   // right child
        y.diff --              // right branch increased
        if y.diff ≥ 0      // break because left branch is ≤0 right branch
            break
    z = y
    y = y.parent
```

The original procedure is modified by adding a single while loop that starts at the newly inserted node, and loops up the tree, performing a constant # of iterations at each iteration. Therefore, the runtime is still $O(h)$.

Q5(a) EarlyBudget(T, s)

```
if T = NIL
    return 0
L = 0
if T.start < s
    L += T.budget
    if T.left ≠ NIL
        L += T.left.btotal
        Return L + EarlyBudget(T.right, s)
else
    Return EarlyBudget(T.left, s)
```

The procedure loops down one path of the tree, performing a constant # of operations at each recursive call. Therefore the runtime is $O(h)$

(b) ProjectsAfter (T, K)

```
m = KeyRank(T, K.end)   // returns # of projects that start on or
Return n-m               before time K.end
```

We showed in class that the rank algorithm runs in time $O(h)$.

(c) PostBudget(T, K)

```
if T = NIL
    return 0
B = 0
if T.start > K.end
    B += 1
    if B.right ≠ NIL
        B += B.right.btotal
    Return B + PostBudget(T.left, K)
else
    return PostBudget(T.right, K)
```

(d) Use Range(a, b) from week 8, which runs in time $O(h)$, OR:

```
O(h) { m1 = KeyRank(T, a)
     { m2 = KeyRank(T, b)
Return m2 - m1        // Note this option does not include
                         projects which start at time a
```

Changed to z, to match in class notation

e) ProjectInsert(T, z)

```
z.size = 1
z.max = z.int.high
z.btotal = z.budget
if T = NIL
    return z
else x = T
    while x ≠ NIL
        x.size += 1
        x.btotal += z.budget
        if z.max > x.max
            x.max = z.max
        y = x
        if z.start < x.start
            x = x.left
        else x = x.right
    z.parent = y
    if z.start < y.start
        y.left = z
    else y.right = z
```

Newly added code, which updates attributes.
Constant-time change per iteration of while loop. ∴ O(h)